Discrete mathematics

Petr Kovář & Tereza Kovářová
petr.kovar@vsb.cz

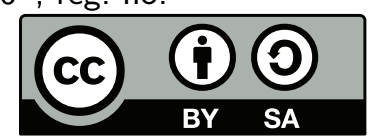
VŠB – Technical University of Ostrava

Winter Term 2022/2023
DiM 470-2301/02, 470-2301/04, 470-2301/06

EUROPEAN UNION
European Structural and Investment Funds
Operational Programme Research,
Development and Education

MINISTRY OF EDUCATION,
YOUTH AND SPORTS

The translation was co-financed by the European Union and the Ministry of Education, Youth and Sports from the Operational Programme Research, Development and Education, project "Technology for the Future 2.0", reg. no. CZ.02.2.69/0.0/0.0/18_058/0010212.



About this file

This file is meant to be a guideline for the lecturer. Many important pieces of information are not in this file, they are to be delivered in the lecture: said, shown or drawn on board. The file is made available with the hope students will easier catch up with lectures they missed.

For study the following resources are better suitable:

- Meyer: Lecture notes and readings for an http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-042j-mathematics-for-computer-science-fall-2005/readings/" (weeks 1-5, 8-10, 12-13), MIT, 2005.
- Diestel: Graph theory http://diestel-graph-theory.com/ (chapters 1-6), Springer, 2010.

See also http://homel.vsb.cz/~kov16/predmety_dm.php

Lecture overview

Algorithms for discrete structures

- implementing basic structures
- implementing sets
- list of all selections or arrangements
- generating random numbers
- combinatorial explosion

7. Algorithms for discrete structures

In this chapter we describe how to implement some structures and algorithms introduced in Part I.

Some structures are easy to implement, some require a rather elaborate approach. They often differ in memory requirements or CPU time requirements.

Usually a general approach is more time/space consuming, on the other hand the generality must not necessarily be payed for by significantly slower algorithm.

This chapter is dedicated to selected implementations of structures and algorithms.

7.1. Implementing basic structures

- sequences
- mappings
- relations
- permutations

A (finite) sequence $(a_0, a_1, \ldots, a_{n-1})$
we implement as a one-dimensional field `a[ ]`, where `a[i]` = a_i.

Example

We have a (finite) sequence $(7, 5, 5, 7, 5, 6, 6)$.
We store the sequence in an array `p = [ 7 5 5 7 5 6 6 ]`.

Mappings

Mapping $f : A \to B$

Let us take a finite $A = \{a_0, a_1, \ldots, a_{n-1}\}$ and $B = \{b_0, b_1, \ldots, b_{m-1}\}$. Wework with subscripts only and implement the mapping as a sequence – field `f[ ]`, in which `f[i]=j` stands for $f(a_i) = b_j$.

This is particularly suitable when A and B are integer sets with small integers. For different sets we have to "translate" elements in A, B into their indices (usually CPU time consuming).

- for elements in A we can use hash tables
- for elements in B we use structured data types or pointers

Example

We have a mapping $f : [0, 5] \to [0, 5]$, where $f(0) = 4$, $f(1) = 5$, $f(2) = 3$, $f(3) = 3$, $f(4) = 2$, $f(5) = 2$.
We store the mapping in a field `f = [ 4 5 3 3 2 2 ]`.

Example

Take a mapping $f : \{A, B, C, D, E\} \to \{x, y, z, w\}$,
where $f(A) = w$, $f(B) = z$, $f(C) = w$, $f(D) = x$, $f(E) = w$.
Mapping is stored in a field `f = [ 3 2 3 0 3 ]`.
Potřebujeme pomocná pole `X = [ A B C D E ]`, `Y = [ x y z w ]`.

Example

Take a mapping $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, where $f(x, y) = x^2 + 3y$.
Cannot be stored in a field! Not possible to store $\mathbb{R}$.

Example

Take a mapping $f : [-5 : 5] \times [-5 : 5] \to \mathbb{R}$, where $f(x, y) = x^2 + 3y$.
Mapping is stored in a two-dimensional field with 11×11 (approximate?) values.

Example

Take a mapping $f : [-5 : 5] \times [-5 : 5] \to \mathbb{R}$, where $f(x, y) = \sqrt{x + y}$.
Mapping is stored in a two-dimensional field with 11×11 (approximate!) values.

Binary relations

Binary relation R on the set A

For finite and small $A = \{a_0, a_1, \ldots, a_{n-1}\}$ we implement relation by a two-dimensional field (matrix) `r[ ][ ]`, in which
`r[i][j] = 0` when $(a_i, a_j) \notin R$ and
`r[i][j] = 1` when $(a_i, a_j) \in R$.

Example

We have a relation $R \subseteq [0,4]^2$, where $R = \{(0,0),(0,4),(1,3)(2,4)\}$.
Relation R can be stored in a two-dimensional field

$$\mathtt{R} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Properties of relations

Take a binary relation R on the set $\{0, 1, \ldots, n-1\}$ given by the field `r[][]`.

Check if the relation `r` is reflexive, $O(n)$

```
for (i=0; i<n; i++)
   if (!r[i][i]) {    // all ones?
      printf("Not reflexive!");
      return -1;
   }
```

Check if the relation `r` is symmetric, $O(n^2)$

```
for (i=0; i<n; i++)
   for (j=i+1; j<n; j++)
      if (r[i][j]!=r[j][i]) {     // a symmetric matrix?
         printf("Not symmetric!");
         return -1;
      }
```

Properties of relations (continued)

Is the relation `r` transitive? Verify for each tripple

$$\forall i, j, k : \; r[i][j] \; \wedge \; r[j][k] \; \Rightarrow \; r[i][k].$$

Check if the relation `r` is transitive, $O(n^3)$

```
for (i=0; i<n; i++)
   for (j=0; j<n; j++) {
      if (!r[i][j]) continue;          // skip
      for (k=0; k<n; k++) {
         if (!r[j][k]) continue;       // skip
         if (r[i][k]) continue;        // has to be!
         printf("Not transitive!");
         return -1;
      }
   }
```

Example

We have a relation $R \subseteq [0,4]^2$, where $R = \{(0,0),(0,4),(1,3)(2,4)\}$ stored in a two-dimensional field

$$\mathtt{R} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Relation is not reflexive.
Relation is not symmetric.
Relation IS transitive.

Question

How to test antisymmetry?
How to test linearity?
What is the time-complexity of these test?

Permutations

A permutation we implement as a bijective mapping $p : [0, n-1] \to [0, n-1]$.

Example

We have a permutation $\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 2 & 1 & 3 & 0 & 5 \end{pmatrix}$, π can be stored in a field

$$\mathtt{p} = [4\ 2\ 1\ 3\ 0\ 5]$$

How to verify, the one dimensional field describes a permutation? It is enough to verify if p is onto (surjective).

Check whether p[ ] is a permutation, $O(n)$

```
for (i=0; i<n; i++)  u[i] = 0;          // an auxiliary field
for (i=0; i<n; i++)  if (p[i]>=0 && p[i]<n)  u[p[i]] = 1
   else printf("Not a permutation!");     // out of range
for (i=0; i<n; i++)
   if (u[i]!=1)
      printf("Not a permutation!");
```

Permutations (continued)

Composition of `p[ ]` and `q[ ]` is the permutation `r[ ]`, $O(n)$

```
for (i=0; i<n; i++)
     r[i] = q[p[i]];
```

We can obtain a list of all cycles by the code:

Cycle notation of an `n`-element permutation `p[ ]` of $[0, n-1]$, $O(n)$

```
for (i=0; i<n; i++)  u[i] = 0;          // an auxilliary field
for (i=0; i<n; i++) if (u[i]==0) {      // not used
   printf("\n(%d",i);  u[i] = 1;        // start cycle
   for (j=p[i]; j!=i; j=p[j])  {        // next in this cycle
      printf(",%d",j);  u[j] = 1;
   }
   printf(")");                         // close cycle
}
```

7.2. Set implementation

Set are not easy to implement. The problems include

- search for a particular element (not a specified list index),
- guarantee non-repetitive elements.

Characteristic function of a subset

The *universe* $\mathcal{U} = \{u_0, u_1, \ldots, u_{n-1}\}$, from which elements are taken, has to be known. Subsets $X \subseteq \mathcal{U}$ are implemented as fields `x[ ]`, where

$$\texttt{x[i]} = \begin{cases} 1 & \text{for } u_i \in X \\ 0 & \text{otherwise.} \end{cases}$$

Advantages: easy to search for a particular element, unions by using the *OR* function, intersection by *AND* function.

Disadvantage: suitable only for small universe $\mathcal{U}$!

Example

The set $A = \{2, 3, 5\}$ in a universe $U = [1, 10]$ is implemented using a characteristic function in a field A = [0 1 1 0 1 0 0 0 0 0].

Example

The set $A = \{2, 3, 5\}$ in a universe $U = [1, 100]$ is implemented using a characteristic function in a field A = [0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0].
The set $B = \{1\}$, B = [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0].

To find the union $A \cup B$ on U use binary OR. To find the intersection $A \cap B$ use binary AND.

Example

The set $A = \{2, 3, 5\}$ in the universe $U = \mathbb{N}$ cannot be implemented.

List of elements

The set X is implemented as a list of elements. The list of k elements in X is stored in a field `x[ ]`, we can write

$$X = \{\mathtt{x}[1], \mathtt{x}[2], \ldots, \mathtt{x}[\mathtt{k}]\} \text{ for the field } \mathtt{x}[\ ] \text{ of length } k.$$

Advantage: work for big or even unspecified universe.

Instead of a field one can use a *dynamic linked list*, then it is easy to add or remove elements from the list.

Disadvantage: determining if a particular object is in the set (often used operation) is costly – one has to go through the entire list.

Example

The set $A = \{2, 3, 5\}$ in the universe $U = [-MAX\_INT, MAX\_INT]$ is implemeted as a field A = [2, 3, 5].

Example

The set $A = \{3, 5, 2\}$ in the universe $U = [-MAX\_INT, MAX\_INT]$ is implemeted as a field A = [3, 5, 2].

Test if element x is in the set a[ ] of size n, $O(n)$

```
for (i=0; i<n; i++) {              // traverse all field a[ ]
   if (a[i]==x) break;             // is x in a[ ]?
}
if (i<n) printf("Element x is in field a[ ]");   // found?
```

Union of two set in fields a[ ],b[ ] into the field c[ ], $O(n^2)$

```
for (i=0; i<m; i++)
   c[i] = a[i];                    // all m elements from a[ ]
for (i=0,k=m; i<n; i++) {          // next n elements from b[ ]
   for (j=0; j<m; j++)
      if (b[i]==a[j]) break;       // if b[i] in a[ ]
   if (j<m) continue;              // skip
   c[k++] = b[i];                  // or add it to c[ ]
}
```

Ordered list of elements

An easy modification of the previous implementation.
The elements in the list are ordered according some rule (length, size, lexicographic, etc.)

Advantage: on can use *binary search* in the set of elements by bisection in the list (see Example).

Example

The set $A = \{2, 3, 5\}$ in the universe $U = [-MAX\_INT, MAX\_INT]$ is implemented as a field A = [2, 3, 5].

Example

The set $A = \{3, 5, 2\}$ in the universe $U = [-MAX\_INT, MAX\_INT]$ is implemented as a field A = [2, 3, 5].

Binary search for k in an ordered field `p[ ]` of length `n`

```
int a = 0; b = n-1;
while (a<b && p[a]!=k) {          // k found?
   c = (a+b)/2;
   if (p[c]<k)  a = c+1;          // no, it will be bigger
   else         b = c;            // no, it will be smaller
}
if (p[a]!=k)  printf("The number k not in the list.");
```

Just $\lceil \log_2 n \rceil$ searching steps.

Adding a new element `x` to the set in a field `a[ ]` requires $O(n)$ operations:

- find the proper place, $O(\lceil \log_2 n \rceil)$
- copy or "shift" part of the field, $O(n)$

Similarly, when removing elements.

Union of two ordered list in fields a[ ], b[ ] of size m, n into an ordered field c[ ] of size l, $O(n+m)$

```
int i=0, j=0, k, l=0;
for (k=0; k < m+n; k++) {
   if (i >= m) {                    // if a[ ] exhausted
      c[l++] = b[j++];
      continue;
   }
   if (j >= n) {                    // if b[ ] exhausted
      c[l++] = a[i++];
      continue;
   }
   if (a[i] == b[j]) {              // just one copy
      j++;
      continue;
   }
   c[l++] = (a[i] < b[j]) ? a[i++] : b[j++];
}
```

Summary

- How large is the universe?
- Will we (and how often) modify the structure of the set?
- Will we (and how often) search the set?
- Will we (and how often) construct unions of sets?

... pick the approriate model.

7.3. Listing selection and arrangements

Often we have to traverse all selections or arrangements of a given type:

- different mappings,
- k-permutations,
- k-combinations without repetitions.

Simple traversing of all ordered pairs (triples, etc.)

All ordered pairs of indices `i,j` we traverse in a nested loop

2-permuations with repetition, $O(n^2)$

```
for (i=0; i<n; i++)                    // nested loop
   for (j=0; j<n; j++) {
      // process a particular ordered pair (i,j)
   }
```

All unordered pairs of indices `i,j` are traversed similarly

2-combinations, $O(n^2)$

```
for (i=0; i<n; i++)                 // just "above the diagonal"
   for (j=i+1; j<n; j++) {
      // process a particular unordered pair {i,j}
   }
```

Processing all permutations of an n-element set A in a[ ]

All $n!$ permutations are processed by a recursive algorithm (Heap 1963).

Heap's algorithm – permutations of n elements

```
int i, a[ ];
permutation(n, a[ ]) {
   if (n==1)
      // process permutation in a[ ]
   else
      for (i=0; i < n-1; i++) {
         permutation(n-1, a[ ]);
         if (n even)
            swap(a[i], a[n-1]);
         else
            swap(a[0], a[n-1]);
      }
      permutation(n-1, a[ ]);
}
```

Function `swap(x,y)` simply swaps the content of `x` and `y`.

Processing all mappings

All n^k mapping of a k-element set into and n-element set

$$\texttt{map} : \{0, 1, \ldots, k-1\} \rightarrow \{0, 1, \ldots, n-1\}$$

we traverse by the following code: k nested cycles not necessary!

k-permutations with repetition of n elements, $O(n^k)$

```
int i, map[k];
map[i = 0] = -1;
while (i>=0) {
   if (++map[i]>=n)              // increase by 1
     { i--; continue; }
   if (++i<k)                    // 'erase' next element
     { map[i] = -1; continue; }
   // process the mapping (map[0], ..., map[k-1])
   i--;
}
```

For each choice we verify

- if it exceeded n, then we return to the previous level,
- if this was the last choice for the k-th element, otherwise next level.

Processing all k-permutations (without repetitions) on n elements

k-permutations without repetition of n elements, $O(n^k)$

```
int i, j, arrange[k];
arrange[i = 0] = -1;
while (i>=0) {
   if (++arrange[i]>=n)        // increase by 1
      { i--; continue; }
   for (j=0; j<i; j++)         // does it repeat?
      if (arrange[i]==arrange[j])  break;
     if (j<i)  continue;       // skip repeated
   if (++i<k)                  // 'erase' next elements
      { arrange[i] = -1; continue; }
   // process k-permutation (arrange[0],...,arrange[k-1])
   i--;
}
```

For each choice we verify

- if it exceeded n, then we return to the previous level,
- if this is not a repeated element, then we skip it,
- if it was the last k, otherwise proceed by the next level.

Processing all k-combiations

We traverse all k-combinations (without repetition) on n elements.
It is similar to the previous case, but now we produce
ordered k-tuples. Hence every k-combination is obtained just once.

k-combinations (without repetition) of n elements, $O(n^k)$

```
int i, select[k];
select[i = 0] = -1;
while (i>=0) {
   if (++select[i]>=n)                  // increase by 1
      { i--; continue; }
   if (++i<k) {
      select[i] = select[i-1];          // sorted already!
      continue;
   }
   // process the k-combinaton (select[0],...,select[k-1])
   i--;
}
```

We do not have to check for repeated selections, since the elements of the selection are ordered.

7.4. Generating random numbers

We investigate really random sequence of bits in a computer.

Where do we require random numbers/bit sequences?

- generating random (large) private keys (for SSL certificates).
  Using random passwords for SSL encryption (if not random, it can be broken!).
- Resolving packet collisions on Ethernet by random pauses before next transmission.
- Used by *probability algorithms*, random bits can boost computation performance.
- In statistical analysis of real events, modelling real chaotic and physical experiments, etc.

Various random number generators

Elemental pseudorandom generators

Use various formulas as

$$x := (A \cdot x + B) \mod C.$$

We iterate this and certain bits x are used as the random sequence.

Disadvantage: *depends heavily* on previous iterations and *predictable*.

Pseudorandom generators with external input

Similar formulas as in the previous case with additional input from *external physical processes* (key press delays, disk reading delays, network statistics, etc.)

Problems: dependence on external conditions, *can be influenced* by the environment, each bit is "costly".

Hardware random generators

Based on *quantum noise* (in semiconductors).

Problems: translation into a uniform bit sequence, confidence in quantum mechanics.

7.5. Combinatorial explosion

In software based solution of problems in discrete mathematics we often require algorithms such as:

Traverse all cases.

Then we may encounter the phenomenon called *exponential combinatorial explosion*.

- fast growth of the factorial.
- tale about corn on chessboard fields

If the number of traversed cases grows exponentially, them even for an input increased by 1 the computational time increases many times.
In many situation in input of size 10 can be solved in seconds on a 386 processor, but the input of size 15 cannot be counted on most powerful machines in the world.

Remember this phenomenon when designing your algorithm with "brute force"!

By choosing an appropriate algorithm, data structure or input limitation we can achieve tremendous increase of performance.

Example

How many (non-isomorphic!) tournaments of n teams (disregards the round order, disregard the team numbers).

n=2 1 tournament

n=4 1 tournament

n=6 1 tournament

n=8 6 tournaments

n=10 396 tournaments

n=12 526 915 620 tournaments

n=14 1 132 835 421 602 062 347 tournaments

n=16 ?

If we distinguish the team numbers, then for $n = 14$ is 98 758 655 816 833 727 741 338 583 040 tournaments.

Next lecture

Part II Introduction to Graph Theory

Chapter 1. The graph

- motivation
- definition of a graph
- degree of a vertex